# Anti-Debug: Exceptions

anti-debug.checkpoint.com/techniques/exceptions.html

## Contents



## Exceptions

The following methods deliberately cause exceptions to verify if the further behavior is not typical for a process running without a debugger.

### 1. UnhandledExceptionFilter()

If an exception occurs and no exception handler is registered (or it is registered but doesn't handle such an exception), the `kernel32!UnhandledExceptionFilter()` function will be called. It is possible to register a custom unhandled exception filter using the `kernel32!SetUnhandledExceptionFilter()`. But if the program is running under a debugger, the custom filter won't be called and the exception will be passed to the debugger. Therefore, if the unhandled exception filter is registered and the control is passed to it, then the process is not running with a debugger.

**x86 Assembly (FASM)**

```
include 'win32ax.inc'

.code

start:
        jmp begin

not_debugged:
        invoke  MessageBox,HWND_DESKTOP,"Not Debugged","",MB_OK
        invoke  ExitProcess,0

begin:
        invoke SetUnhandledExceptionFilter, not_debugged
        int  3
        jmp  being_debugged

being_debugged:
        invoke  MessageBox,HWND_DESKTOP,"Debugged","",MB_OK
        invoke  ExitProcess,0

.end start
```

### C/C++ Code

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo)
{
    PCONTEXT ctx = pExceptionInfo->ContextRecord;
    ctx->Eip += 3; // Skip \xCC\xEB\x??
    return EXCEPTION_CONTINUE_EXECUTION;
}

bool Check()
{
    bool bDebugged = true;

SetUnhandledExceptionFilter((LPTOP_LEVEL_EXCEPTION_FILTER)UnhandledExceptionFilter);
    __asm
    {
        int 3                       // CC
        jmp near being_debugged     // EB ??
    }
    bDebugged = false;

being_debugged:
    return bDebugged;
}
```

## 2. RaiseException()

Exceptions such as `DBC_CONTROL_C` or `DBG_RIPEVENT` are not passed to exception handlers of the current process and are consumed by a debugger. This lets us register an exception handler, raise these exceptions using the `kernel32!RaiseException()` function, and check whether the control is passed to our handler. If the exception handler is not called, the process is likely under debugging.

**C/C++ Code**

```
bool Check()
{
    __try
    {
        RaiseException(DBG_CONTROL_C, 0, 0, NULL);
        return true;
    }
    __except(DBG_CONTROL_C == GetExceptionCode()
        ? EXCEPTION_EXECUTE_HANDLER
        : EXCEPTION_CONTINUE_SEARCH)
    {
        return false;
    }
}
```

## 3. Hiding Control Flow with Exception Handlers

This approach does not check whether a debugger is present, but it helps to hide the control flow of the program in the sequence of exception handlers.

We can register an exception handler (structured or vectored) which raises another exception which is passed to the next handler which raises the next exception, and so on. Finally, the sequence of handlers should lead to the procedure that we wanted to hide.

Using Structured Exception Handlers:

**C/C++ Code**

```
#include <Windows.h>

void MaliciousEntry()
{
    // ...
}

void Trampoline2()
{
    __try
    {
        __asm int 3;
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        MaliciousEntry();
    }
}

void Trampoline1()
{
    __try
    {
        __asm int 3;
    }
    __except (EXCEPTION_EXECUTE_HANDLER)
    {
        Trampoline2();
    }
}

int main(void)
{
    __try
    {
        __asm int 3;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {}
    {
        Trampoline1();
    }

    return 0;
}
```

Using Vectored Exception Handlers:

### C/C++ Code

```
#include <Windows.h>

PVOID g_pLastVeh = nullptr;

void MaliciousEntry()
{
    // ...
}

LONG WINAPI ExeptionHandler2(PEXCEPTION_POINTERS pExceptionInfo)
{
    MaliciousEntry();
    ExitProcess(0);
}

LONG WINAPI ExeptionHandler1(PEXCEPTION_POINTERS pExceptionInfo)
{
    if (g_pLastVeh)
    {
        RemoveVectoredExceptionHandler(g_pLastVeh);
        g_pLastVeh = AddVectoredExceptionHandler(TRUE, ExeptionHandler2);
        if (g_pLastVeh)
            __asm int 3;
    }
    ExitProcess(0);
}


int main(void)
{
    g_pLastVeh = AddVectoredExceptionHandler(TRUE, ExeptionHandler1);
    if (g_pLastVeh)
        __asm int 3;

    return 0;
}
```

## Mitigations

- During debugging:
  - For debugger detection checks: Just fill the corresponding check with `NOP`s.
  - For Control Flow hiding: You have to manually trace the program till the payload.
- For anti-anti-debug tool development: The issue with these type of techniques is that different debuggers consume different exceptions and do not return them to the debugger. This means that you have to implement a plugin for a specific debugger and change the behavior of the event handlers which are triggered after the corresponding exceptions.